MODIS
Software Development Standards and Guidelines

Version 1

October 25, 1995

SDST-022

MODIS
Software Development Standards and Guidelines

Reviewed By:

_____

James Firestone, General Sciences Corporation/SAIC Date
SDST STIG Lead

_____

Jenny Glenn, General Sciences Corporation/SAIC Date
SDST Systems Analyst

_____

Barbara Putney, GSFC/Code 920 Date
MODIS Systems Engineer

_____

Catherine Harnden, GSFC/Code 920 Date
MODIS SDST Deputy Manager

_____

Laurie Schneider, General Sciences Corporation/SAIC Date
SDST R&QA Manager

_____

Tom Piper, General Sciences Corporation/SAIC Date
MODIS Task Leader

Approved By:

_____

Edward J. Masuoka, GSFC/Code 920 Date
MODIS SDST Manager

Change Record Page

This document is baselined and has been placed under Configuration Control. Any changes to this document will need the approval of the Configuration Control Board.

Document Title: MODIS Software Development Standards and Guidelines

Document Date: October 25, 1995

Issue	Date	Page Affected	Description
Original	2/15/1995	All	Baseline Review
Change 1	10/25/95		CCRs: 026, -049

MODIS
Software Development Standards and Guidelines

Table of Contents

MODIS
Software Development Standards and Guidelines

1. ~~1.~~ INTRODUCTION

1.1 Purpose and Scope

This document defines and describes the standards and guidelines to be used for developing and maintaining software for use at the MODIS Team Leader Computing Facility (TLCF) and for delivery to the EOSDIS Project. Included are the tools and environment to be used at the SCFs, documentation standards, programming standards, and integration standards.

The standards and guidelines that are in this document are intended to facilitate the integration of the software into the PGS, to make the software more easily maintainable in the future, and to assist in the portability of the code. A standard must be followed. A guideline is a recommended practice.

1.2 Relationship to Project Standards

This document contains in Appendix A the EOSDIS document 423-16-01, Data Production Software and Science Computing Facility (SCF) Standards and Guidelines, which will be referred to as the Project Standards and Guidelines (PSG). This document is inclusive of or supersedes all the standards and most of the guidelines in the PSG. For easy reference, if there is a relationship between an item in this document and the PSG, it is noted in the text.

1.3 Waivers

When the project waiver policy is specified, we will adopt it for any item that does not meet the project standards. Requests for waivers to any of the standards should be sent to the Algorithm Transfer Team member who is integrating the algorithm.

1.4 Document Status and Updates

This document is expected to change as the PSG document changes. The Apr. 1994 DPFTS meeting approved a number of changes to the PSG. Those approved changes need to be approved by the Project before they become standards. Two of those changes have already been incorporated into this document because they are significant to the beta software development. They are:

1. ANSI FORTRAN 90 has been approved (see Sections 7 and 8) and

2. Script file languages have been approved (see Section 11).

All changes will be incorporated in this document once they are approved by the Project. In addition, a number of sections may have TBD items that we plan to define in future versions. The FORTRAN 90 guidelines are expected to change as we become more familiar with it.

2. SCF STANDARDS AND GUIDELINES

All SCF Standards and Guidelines are accepted as they are stated in section 2 of the PSG with the exception of 2.3.2.1-f which is superseded by our documentation standards in Section 3.0. Refer to Appendix A, Section 2 of the PSG for these standards and guidelines.

3. DOCUMENTATION STANDARDS

Documentation that is to be delivered from the SCF to the TLCF may be done in Word, WordPerfect, or ASCII. Documentation from the TLCF to the Project will be done in Word or WordPerfect. ASCII will be used for packing lists and similar text files. (Supersedes PSG 2.3.2.1-f.)

4. PROGRAMMING STANDARDS AND GUIDELINES

This section begins with standards and guidelines for commenting code. Following that are standards and guidelines for C, FORTRAN 90, and FORTRAN 77 coding. It is not believed that any of the MODIS code will be done in Ada so no standards or guidelines were included for that language. Although we have included FORTRAN 77, the preference is that coding be done in FORTRAN 90. This section concludes with script standards and guidelines.

4.1 Commenting Standards

The commenting standards consist of the standard module prolog and a set of guidelines for documenting the rest of the code.

4.1.1 Module Identification Standard

4.1.1.1 Source code (Supersedes PSG 3.4.2.)

To allow identification of individual items of code, a header (prolog), as defined below, shall be inserted at the top of each module in the production software. A module is a main program, subroutine, procedure, function, etc. This header should also be included at the top of insert/include files, with the exception that the blocks relating to input and output parameters are omitted. Prologs are required by the ~~Note: The example uses the C language style of comments, and would need to be converted for other languages. All entries preceded with "!" are mandatory.~~MODIS coding standards to facilitate the integration of the science code. The System Transfer and Integration

Groups will rely heavily on information in the prologs to direct them in the integration of the SDP Toolkit.

NOTE: The example uses the C language style of comments and would need to be converted for other languages. An exclamation mark "!" is used in the following example to indicate all sections of the prolog that are mandatory, and the line numbers are to be used in conjunction with the descriptions that follow. Line numbers and "!" are not required in the actual code.

For heritage code, where the generation of this header information for every module is an unreasonable burden, an alternative approach based on compilation units can be used. A single header can be used to record the description and version history for all modules contained within a file which is compiled as a unit. In this case the only requirement for each module is to describe each input/output parameter (not globals). Although this alternative approach is acceptable for heritage code, the former approach, with a header for each module, is required for all new code.

The inclusion of the headers will allow automated tools within the DAAC I&T environment to manipulate the software items and will ease understanding by the I&T team.

Templates for module headers and include file headers can be accessed through the PGS Toolkit interface software.

```
line no.
00   geonav(float pos[3],float smat[3][3],float coef[6],float sun[3],
01   int nsta,int ninc,int npix,float xlat[409],float xlon[409],
02   float solz[409],float sola[409],float senz[409],float sena[409])
03/*
04!C************************************************************
05
06 !Description:    This subroutine calculates the sensor
07  orientation from the orbit position vector and input values of
08  attitude offset angles.  The calculations assume that the angles
09  represent the yaw, roll and pitch offsets between the local
10  vertical reference frame (at the spacecraft position) and the
11  sensor frame. The outputs are the matrix which represents
12  the transformation from the geocentric rotating to sensor
13  frame, and the coefficients which represent the Earth scan
14  track in the sensor frame. The reference ellipsoid uses an
15  equatorial radius of 6378.137 km and a flattening factor of
16  1/298.257 (WGS 1984).05
17
18 !Input Parameters:
19  pos[3]          satellite position
20  smat[3][3]   sensor orientation matrix
21  coef[6]         scan line coefficients
22  sun[3]          unit sun vector in geocentric rotating coordinates
23  nsta             number of first pixel to start with
24  ninc            increment between pixels for computations
25  npix            number of pixels in scan line
26
27 !Output Parameters:
28  xlat[409]       latitude values
29  xlon[409]     longitude values
```

```
30  solz[409]      solar zenith values
31  sola[409]      solar azimuth values
32  senz[409]     sensor zenith values
33  sena[409]      sensor azimuth values
34
35 !Revision History:
36  $LOG: geonav.c,v $
37  Revision 01.01  1993/10/12 10:12:28
38  Z. GREEN (zgreen@harp.gsfc.nasa.gov)
39  Initial delivery of software.  Modified to comply with ESDIS
40  standards. It was a breeze.
41
42 Revision 01.00  1993/04/29 17:12:28
43 A. SMITH (asmith@harp.gsfc.nasa.gov)
44 Initial debugged version, based on original FORTRAN 77
45 subroutine developed in 1983 by C. ADAMS of the TELLUS
46 project.
47
48 !Team-unique Header:
49  ! References and Credits
49a ! Design Notes
50 !END*********************************************
    */
                                                  <code follows here>
```

The following notes describe how to use the header block, using the above example as a reference.

Line 00: Name of main procedure, include file or subroutine/ procedure/ function. If this is not the main procedure or an include file, it should contain the function statement.

Line 04: Start of prolog. Initial marker can take the following values:

!FXY - contains FORTRANXY (XY = 77 or 90) executable statements

!C - contains C executable statements

!ADA - contains Ada executable statements

!FXY-INC - FORTRANXY (XY = 77 or 90) include file

!C-INC - C include file

Line 06: A concise but complete summary of the overall function of the module. Any references for methods and/or algorithms should be included. Use as many lines as necessary.

Line 18: Header for input parameters

Line 19-25: Input parameters (not global variables) in the order they are presented to the module with a short 1-2 line description of the parameter (and its units where appropriate). Global variables should be described in the module in which they are declared (i.e., only once).

Line 27: Header for output parameters.

Line 28-33: Output parameters (not global variables) in the order they are contained in the function statement. Global variables should be described in the module in which they are declared (i.e., only once). Same format as for input parameters.

Line 35: Start of Revision History Log.

Line 36: If you are using an automated tool for revision control, you should insert any statements required immediately after the Modification History Log Header.

Line 37-46: Each revision should contain as a minimum the revision number, date, time, person, and email address, with a short description of ALL the changes made. The first revision should include the original author of the code. Revisions should be ordered with the latest first. [Note: this revision information can be supplemented with more detailed comments in the code referencing the revision number].

Use a revision numbering scheme where the revision number is in the format "nn.mm" where "mm" is updated each time a change is made to the module and "nn" is updated when the function of the module changes or the algorithm/method is changed. [Note: The release number for the data production software is not related to the revision numbers on individual modules - the release number scheme should be determined by the development team. The date associated with a release is more meaningful than a release number - to those outside the development team.]

Line 48: Start of Team-unique Header. For Version 1 code, SDST will define a standard MODIS Team header that will appear in all prologs. This is required by the ESDIS for the software released to the DAAC.

Line 49: Any references and credits should be noted here.

Line 49a: Any design notes, limitations, etc. should be noted here.

Line 50: End of source code prolog.

4.1.1.2 Script file A prologue similar to the one for source code will be used.

4.1.2 Commenting Guidelines

All code should have enough comments to make it understandable to another programmer. The following are some guidelines to follow as you comment the code.

4.1.2.1 Comments should be maintained to ensure their correctness.

4.1.2.2 Comments should account for a significant percentage of the lines in the source code.

4.1.2.3 Comments should appear before every major control construct (loops, if-thens, switches, etc.)

4.1.2.4 Above-the-line comments should be immediately above the line of code without separating; white space should be indented to the same column as the line of code.

4.1.2.5 Right margin comments should use enough white space to clearly separate comments from code and should all begin in the same column if they are commenting a block of statements.

4.1.2.6 Comment delimiting should be done in a consistent manner.

5. C CODING STANDARDS

The mandatory coding standards for code generated in the C language follow.

5.1 General Standards

5.1.1 All programs including Science Team Member-supplied programs, shall return an exit code of 0 for success and 1 for failure. The rationale for this standard is maintainability.

5.1.2 The source code shall comply with the ANSI standard specification for C (ANSI/X3.159-1989; C Programming Language Standard). Vendor specific extensions to the standard shall not be used. (Same as PSG 3.1.2.1)

5.1.3 External calls from the operational data production software in the PGS, for system and resource accesses, file I/O requests, error message transaction, and metadata formatting, shall be made through PGS Toolkit calls. (Same as PSG 3.1.2.2)

~~5.0.1.3 The code will be compiled with the ANSI checking option on. (Supersedes PSG 3.1.3.1.)~~

5.1.4 File Inclusion. <> and "" notation will be used for including standard C header files and programmer created header files, respectively. The rationale for this standard is maintainability. (Supersedes PSG Guideline 3.1.3.2)

5.1.5 The correct functioning of code should not depend on the rounding behavior of converting a long double to other floating types or double to a float. (Same as PSG 3.1.3.17.)

5.1.6 Float and double variables should not be compared for strict equality (i.e., using == or !=). (Same as PSG 3.1.3.13.)

5.1.7 Do not use pragma preprocessor directives. Different compilers may produce different implementations of these directives, or (if the directive is not recognized) no implementation at all. The rationale for this standard is portaability.

5.1.8 The div and ldiv functions in the C Standard Library will be used rather than the % operator to obtain consistent values of remainders when the quotient is negative. (Supersedes PSG Guideline 3.1.3.16.) The rationale for this standard is maintainability.

5.1.9 The source code shall comply with IEEE Standard 1003.1, POSIX-Part 1: System Application Program Interface (API) [C Language] and with the restrictions found in Appendix D of the PGS Toolkit Users Guide for the ECS Project. (Same as PSG 3.1.2.3)

5.2 Declarations

5.2.1 All variables will be initialized prior to use, i.e., no assumptions should be made that variables are initialized to zero. (Same as PSG 3.1.3.3.)

5.2.2 Shared global definitions will be put in a header file. The rationale for this standard is maintainability.

5.2.3 System library functions will not be declared; the appropriate header files will be included.

5.3 Structuring/Style

5.3.1 Loop control variables will be of integer type. Note: This includes the subtypes of short, long, etc. (Supersedes PSG Guideline 3.1.3.6.) The rationale for this standard is portability.

5.3.2 ~~Unconditional~~ Bbranching (GOTO) will not be used. Exceptions will be made for code inherited from other projects. (Supersedes PSG Guideline 3.1.3.7.)

~~*5.0.3.3* Use a consistent style to highlight code structure and increase readability. This includes a consistent and descriptive naming convention for variables. (Supersedes PSG 3.1.3.5. and 3.1.3.8.) The following will be observed:~~

~~1. All structures will use a consistent indentation scheme within a module for each succeeding level of the structures.~~

~~1. Names of symbolic constants and user-defined macros in #define statements must be in upper-case.~~

5.4 Bit Manipulation

5.4.1 Bitwise operations shall not be used on signed numbers. The rationale for this standard is portability. ~~(Same as PSG 3.1.2.3)~~

5.4.2 Structures will not contain bit field elements. Compilers differ in whether bits are counted from left to right, or from right to left. The rationale for this standard is portability.

5.5 Operators

5.5.1 Use explicit type casts. The rationale for this standard is portability. (Supersedes PSG Guideline 3.1.3.12.) For example:

```
long integer_variable;
long * integer_pointer;
double floating_point_var;
integer_pointer = &integer_variable; /* per guideline */
integer_pointer = &floating_point_var; /* contrary to guideline */
integer_variable = (long)floating_point_var;/* per guideline */
```

5.6 Functions

5.6.1 All user-defined functions will be typed and prototyped. Functions which do not return a value will be typed as void. (Supersedes PSG 3.1.3.14 and 3.1.3.15.) The rationale for this standard is portability. An example of prototyping follows:

```
int GreatestCommonDenominator(int large_term; int small_term);
```

5.7 Error Checking

5.7.1 All return call statuses will be checked and the appropriate actions taken. The rationale for this standard is maintainability.

5.7.2 Main programs will return status of 0 (success) or 1 (fail). The rationale for this standard is maintainability.

6. C CODING GUIDELINES

The following are suggested guidelines for C code. The book C Programming Guidelines by Thomas Plum is recommended as a resource for additional guidelines for writing portable, maintainable, and effective code.

6.1 General Guidelines

6.1.1 Only static variables are guaranteed to retain their value between module calls; all other variables should be assumed to be undefined for each access of a module. (Same as PSG 3.1.3.10.)

~~5.1.1.2 The div and ldiv functions in the C Standard Library will be used rather than the % operator to obtain consistent values of remainders when the quotient is negative. (Supersedes PSG 3.1.3.16.)~~

6.1.23 Free resources that have been allocated.

6.1.34 Use the size–of operator to determine the length of a structure.

6.1.4 It is strongly recommended that the code be compiled with the ANSI checking option on. The rationale for this guideline is portability. (Same as PSG 3.1.3.1.)

6.2 Declarations

6.2.1 Declarations should be ordered consistently throughout the code. Use whatever method makes the most sense for the particular application. Pointers should immediately follow what they are pointing to. For example you could order them as follows:

1. . declaration of external variables in type order
2. . declaration of local variables in type order
3. . declaration of functions used in type order.

The type order could be char, short, unsigned short, long, unsigned long, float, double, enum, struct, union, void. (Supersedes PSG Guideline 3.1.3.4)

6.2.2 All pointer variables should be named in a consistent fashion. For example all pointer variables may consist of the name of the variable with an "_p" at the end (e.g., precip_p may be a pointer to precip).

6.2.3 A pointer should have the same type as the variable it points to. (Same as PSG 3.1.3.9.)

6.3 Structuring/Style

6.3.1 There should be sufficient white space to make the code readable. Blank lines between each major code structure are encouraged.

6.3.2 Use a consistent style to highlight code structure and increase readability. This includes a consistent and descriptive naming convention for variables. All structures will use a consistent indentation scheme within a module for each succeeding level of the structures.

6.3.3 Names of symbolic constants and user-defined macros in #define statements should be in upper-case. All other names should not be in all upper-case. Use of mixed case is encouraged.

~~5.1.4 Header Files~~

~~5.1.4.1 Only the required header files should be included in a particular module. Note that different systems may require different standard header files to be included. Removal of a header file should be done with great care.~~

6.3.4.2 Every header should prevent multiple inclusions of itself.

Example:

/* This is a dummy header file with the name dummy.h that shows how to avoid multiple declarations of variables and functions.

Note that the name of the header file is used as a parameter that is defined or undefined depending on whether it has been invoked in the module processing.

The use of underlines and captial letters make it unlikely that this parameter will be confused with others in the program.

*/

#ifndef __DUMMY_H_ /* see if this header has been invoked before */

#define __DUMMY_H_ /* if it has not, then define it's "definition variable" so it will not be invoked again by the including module. Then proceed to process the header.*/

/*

header code goes here, including necessary #include

*/

#endif

7. FORTRAN 90 CODING STANDARDS

The mandatory coding standards for code generated in the FORTRAN 90 language follow.

7.1 General Standards

7.1.1 Production source code shall comply with the ANSI standard specification for FORTRAN 90. Vender specific extensions will not be used. (Supersedes PSG 3.2.2.1.) The rationale for this standard is portability.

7.1.2 External calls from the operational data production software in the PGS, for system and resource accesses, file I/O requests, error message transaction, and metadata formatting, shall be made through PGS Toolkit calls. (Same as PSG 3.2.2.2.)

7.1.3 Compiler options will not be used to perform functions that can be handled by the language. For example, SAVE statements will be used to statically allocate local variables rather than a compiler option such as -static on the SGI FORTRAN compiler. The rationale for this standard is portability.

~~6.0.1.4 The code will be compiled with the ANSI checking option on.~~

7.1.45 The correct functioning of code should not depend on the rounding behavior of converting a double precision number to a real. The rationale for this standard is portability.

7.1.56 Double precision and real variables should not be compared for strict equality (i.e., using .EQ., .NE., ==, /=). The rationale for this standard is portability.

7.2 Declarations

7.2.1 All variables will be explicitly declared. The IMPLICIT NONE statement will appear in each routine. The rationale for this standard is maintainability.

7.2.2 All variables will be initialized prior to use. (Supersedes PSG Guideline 3.2.3.2.) The rationale for this standard is portability.

7.2.3 PARAMETER variables will not be redefined. (Supersedes PSG Guideline 3.2.3.4.) The rationale for this standard is maintainability.

7.2.4 COMMON Blocks will be named. The rationale for this standard is readability.

7.2.5 Subscripts are required to equivalence two vectors. For example:

EQUIVALENCE (a(1),b(1))

Note: EQUIVALENCE statements are discouraged.

7.3 Structuring

7.3.1 Loop control variables in DO loops will be of INTEGER type. (Supersedes PSG Guideline 3.2.3.8.) The rationale for this standard is maintainability.

7.3.2 Unconditional branching (GOTO) will not be used. (Supersedes PSG 3.2.3.9) The rationale for this standard is maintainability.

~~6.0.3.3 DO-loops will be terminated with ENDDO. The index should not be modified inside the loop. (Supersedes PSG 3.2.3.11.)~~

7.3.34 Use a consistent style to highlight code structure and increase readability. This includes a consistent and descriptive naming convention for variables. (Supersedes PSG Guidelines 3.2.3.7. and 3.2.3.12.) The rationale for this standard is maintainability. The following will be observed:

1. All structures will use a consistent indentation scheme within a module for each succeeding level of the structures.

7.4 Labeling

7.4.1 A consistent labeling scheme will be used, with labels increasing in value through a module. (Supersedes PSG Guideline 3.2.3.14.) The rationale for this standard is maintainability.

7.5 Functions

7.5.1 Generic intrinsic functions will be used rather than type specific functions. (Supersedes PSG Guideline 3.2.3.15.) The rationale for this standard is portability.

7.6 Error Checking

7.6.1 All programs, including Science Team Member-supplied programs, shall return an exit code of 0 for success and 1 for failure. The rationale for this standard is maintainability.

7.6.2 All return call statuses will be checked and the appropriate actions taken. The rationale for this standard is maintainability.

7.6.3 DO-loops will be terminated with ENDDO. The index should not be modified inside the loop. (Same as PSG 3.2.3.11.)

8. FORTRAN 90 CODING GUIDELINES

The following are suggested guidelines for FORTRAN 90 code. At the current time we have no book to recommend for additional guidelines for writing portable, maintainable, and effective code.

8.1 General Guidelines

8.1.1 Format statements should be collected together, preferably at the end of the routine.

8.1.2 Only SAVEd variables are guaranteed to retain their value between module calls; all other variables should be assumed to be undefined for each access of a module. Except for loop indices, variables should not be re-used within a routine.

8.1.3 Free all resources that have been allocated.

8.2 Declarations

8.2.1 Declarations should be ordered consistently throughout the code. Use whatever method best suits the particular application. For example you could order them as follows:

1. declaration of module arguments in the same order as the argument list
2. declaration of global variables in COMMON (using INCLUDE files)
3. declaration of local PARAMETERs (types and values) in type order
4. declaration of local variable types in type order
5. declaration of user defined function types used in type order
6. EXTERNAL declarations
7. INTRINSIC declarations

The type order could be CHARACTER, LOGICAL, INTEGER, REAL, DOUBLE PRECISION, COMPLEX. (Supersedes PSG 3.2.3.3.)

8.2.2 All pointer variables should be named in a consistent fashion. For example all pointer variables may consist of the name of the variable with an "_p" at the end (e.g., precip_p may be a pointer to precip).

8.2.3 A pointer should not point to variables of types different from that indicated by the pointer declaration.

8.3 Structuring/Style

8.3.1 There should be sufficient white space to make the code readable. Blank lines between each major code structure are encouraged.

8.3.2 Use mixed case rather than all upper case.

8.3.3 Computed and arithmetic GOTOs may be used but are strongly discouraged. (Supersedes PSG 3.2.3.10.)

9. FORTRAN 77 CODING STANDARDS

The mandatory coding standards for code generated in the FORTRAN 77 language follow. Heritage code will be accepted in FORTRAN 77. ~~We urge that new code be written in FORTRAN 90.~~

9.1 General Standards

9.1.1 Production source code shall comply with the ANSI standard specification for FORTRAN 77. The Project approved extensions (see PSG 3.2.3.1) ~~needed for PGS Toolkit calls~~ are also accepted for MODIS. ~~Other Project approved extensions may be accepted via a waiver~~

9.1.2 External calls from the operational data production software in the PGS, for system and resource accesses, file I/O requests, error message transaction, and metadata formatting, shall be made through PGS Toolkit calls. (Same as PSG 3.2.2.2)

9.1.3 Do not use compiler options to perform functions that can be handled by the language. For example, SAVE statements should be used to statically allocate local variables rather than a compiler option such as -static on the SGI FORTRAN compiler. The rationale for this standard is portability.

9.1.4 FORTRAN77 code shall not use PGS Toolkit calls that provide capabilities not found in FORTRAN77 (e.g., dynamic memory allocation). The rationale for this standard is that there will be no FORTRAN77 bindings for this class of PGS Toolkit calls. (Same as PSG 3.2.2.3)

9.1.5 The FORTRAN77 source code shall comply with IEEE Standard 1003.9 POSIX FORTRAN77, Language Interfaces, Part 1: Binding for System Application Program Interface (API) and with the restrictions found in Appendix D of the PGS Toolkit Users Guide for the ECS Project. (Same as PSG 3.2.2.4)

9.2 Declarations

9.2.1 All variables will be explicitly declared. The rationale for this standard is maintainability.

9.2.2 All variables will be initialized prior to use. (Supersedes PSG 3.2.3.2.) The rationale for this standard is portability.

9.2.3 PARAMETER variables will not be redefined. (Supersedes PSG 3.2.3.4.) The rationale for this standard is maintainability.

9.2.4 COMMON Blocks will be named. The rationale for this standard is maintainability.

9.2.5 Subscripts are required to equivalence two vectors. For example:

EQUIVALENCE (a(1),b(1))

Note: EQUIVALENCE statements are discouraged.

9.3 Structuring/Style

9.3.1 Loop control variables in DO loops will be of INTEGER type. (Supersedes PSG 3.2.3.8.) The rationale for this standard is maintainability.

9.3.2 Unconditional branching (GOTO) will not be used. Exceptions will be made for well-structured code that is inherited from other projects. (Supersedes PSG 3.2.3.9) The rationale for this standard is maintainability.

9.3.3 DO-loops will be terminated with CONTINUE. The index should not be modified inside the loop. (Supersedes PSG 3.2.3.11) The rationale for this standard is maintainability.

9.3.4 Use a consistent style to highlight code structure and increase readability. This includes a consistent and descriptive naming convention for variables.

(Supersedes PSG Guidelines 3.2.3.7. and 3.2.3.12.) All structures will use a consistent indentation scheme within a module for each succeeding level of the structures. The rationale for this standard is maintainability.

9.4 Labeling

9.4.1 A consistent labeling scheme will be used, with labels increasing in value through a module. (Supersedes PSG 3.2.3.14.) The rationale for this standard is maintainability.

~~7.0.5 Functions~~

~~7.0.5.1 Generic intrinsic functions will be used rather than type specific functions. (Supersedes PSG 3.2.3.15.)~~

9.5 Error Checking

9.5~~6~~.1 All return call statuses will be checked and the appropriate actions taken. The rationale for this standard is maintainability.

9.5.2 All programs including Science Team Member-supplied programs, shall return an exit code of 0 for success and 1 for failure. The rationale for this standard is maintainability.

10. FORTRAN 77 CODING GUIDELINES

The following are suggested guidelines for FORTRAN 77 code. The book Effective FORTRAN 77 by M. Metcalf is recommended for additional guidelines for writing portable, maintainable, and effective code.

10.1 General Guidelines

10.1.1 Format statements should appear together at one spot in the code.

10.2 Declarations

10.2.1 Declarations should be ordered consistently throughout the code. Use whatever method best suits the particular application. For example you could order them as follows:

1. declaration of module arguments in the same order as the argument list
2. declaration of global variables in COMMON
3. declaration of local PARAMETERs (types and values) in type order
4. declaration of local variable types in type order
5. declaration of user defined function types used in type order

6. EXTERNAL declarations
7. INTRINSIC declarations
8. declaration of DATA values in type order.

The type order could be CHARACTER, LOGICAL, INTEGER, REAL, COMPLEX. (Supersedes PSG 3.2.3.3.)

10.3 Structuring/Style

10.3.1 There should be sufficient white space to make the code readable. Blank comment lines between each major code structure are encouraged.

10.3.2 Computed and arithmetic GOTOs may be used but are strongly discouraged. (Supersedes PSG 3.2.3.10.)

10.3.3 Generic intrinsic functions will be used rather than type specific functions. (Same as PSG Guideline 3.2.3.15.)

11. SCRIPT CODING STANDARDS

Adherance to these standards is mandatory for script command languages that are used to generate the command language portion of the science data production software delivered to the DAACs. The following sections on scripts were taken directly from the PSG document (see Appendix A).

11.1 Standard~~Currently the Project is considering four scripting languages which are csh, tcsh, ksh, and Perl.~~s

Command language code delivered to the DAACs as part of the science data production software shall be written using one or more of the folowing script languages: csh, ksh, perl, or POSIX-compatible shell language.

The POSIX standard is published in IEEE Std 1003.2-1992, IEEE Standard Information Technology, Portable Operating System Interface (POSIX), Part 2: Shell and Utilities.

11.2 Guidelines

Command language code delivered to the DAACs as part of the science data production software from a science team should be written using the smallest possible number of script languages.

The use of compiled programming language should be minimized, in developing science data production software for delivery to the DAACs. The rationae for this guideine is efficiency of software execution and adherence to standards for portabiity. Compiled code executes more efficiency that interpreted code. There are formal national standards for the approved complied programming languages while there is only one formal national standard, POSIX, for a script language.

12. INTEGRATION STANDARDS

12.1 File Naming Conventions

All file names will follow the following conventions:

12.1.1 Each Program or group of programs related to one algorithm will have a unique ~~3 to 6 letter~~ identification code prefaces by the process ID number (e.g., MOD_PR10 code will be implemented in programs beginning with PR10)~~. This code will be determined between the programmer and the CM staff.~~

12.1.2 Each file name will use the above id as the prefix to all file names related to that program including any libraries or executables created by it. (Exception - test data sets)

12.1.3 Each file will use the appropriate suffix as described below:

FORTRAN source files - .f

FORTRAN include files - .inc

C source files - .c

C header files - .h

libraries - .a

Word documents - .doc

WordPerfect documents - .wp

ASCII documents - .txt

Script files and executables will have none per the UNIX conventions.

12.2 Runtime Environment

12.2.1 There will be ~~absolutely~~ no hard coded path names. ~~TBD~~

12.3 Makefiles/Imakefiles

The project has not yet determined whether they will be using make or imake. When that decision has been made, standards for the makefile or imakefile will be written. There will be a set of standard compiler options.

~~(This page intentionally left blank.)~~

APPENDIX A: EOSDIS DOCUMENT 423-16-01

This Appendix contains EOSDIS document 423-16-01, Data Production Software and Science Computing Facility (SCF) Standards and Guidelines. The original document took 32 pages due to its formatting. We have made the following changes in order to reduce the size of the document and make it readable:

1. the top and bottom of the page margins were altered to match the rest of this document.

2. the header/footer distances were adjusted to match this document.

3. the page numbers no longer match the original page numbers due to the above changes

4. the table of contents was removed since it was no longer accurate

5. excessive spacing between paragraphs was adjusted

6. any paragraphs whose point size was inconsistent with the rest of this appendix was adjusted.

423-16-01

Data Production Software and Science Computing Facility (SCF) Standards and Guidelines

January 14, 1994 (CH-01, 2/15/95)

GODDARD SPACE FLIGHT CENTER,
GREENBELT, MD.

Table of Contents

Data Production Software and SCF Standards and Guidelines

1. Introduction.

1.1. Purpose.

The purpose of these standards is, fundamentally, to avoid excess costs over the life cycle of the Data Production Software. The standards contained in this document are motivated by a need for maintainable and portable software.

1.2. Scope.

The standards promulgated in this document apply to networked Science Computing Facilities (SCFs) and data production software that will be delivered for integration into the Earth Observing System (EOS) Product Generation System (PGS) at the Distributed Active Archive Centers (DAACs). The scope of these standards explicitly excludes prototype code or supporting software that may be used in building Data Production Software but that will not be delivered to the PGS.

The standards in this document are mandatory. Standards statements always include the word "shall". (CH01) Guidelines found in this document are not mandatory, but are included as recommendations. The guidelines always include the word "should" or "may" in the statement.

1.3. Authority.

This document is issued under the authority of the Earth Science Data and Information System (ESDIS) Project (the Project). The standards were developed by the ESDISP with the assistance and consensus of the Data Processing Focus Team (DPFT). The DPFT membership includes several representatives from EOS Data Production Software Developers and DAACs.

1.4. Waivers.

The Project may issue waivers to these standards for performance or heritage software on a case by case basis.

1.5. Extensions of Standards.

(CH01)The Project anticipates that these standards will be extended in the future.

2. SCF Standards and Guidelines

2.1. SCF Hardware

2.1.1. Intent.

The intent of the SCF hardware standards is to control the number of versions of the PGS Toolkit that the Project must support for the Data Production Software teams.

2.1.2. Standards

2.1.2.1

SCF hardware shall be capable of running the software included in the SCF Software Standards.

2.1.3. Guidelines

2.1.3.1

SCF hardware hosting the PGS Toolkit should be compatible with hardware from the set of PGS Toolkit hosts established by the ECS Contractor.

2.2. SCF Communications

2.2.1. Intent.

The intent of this standard is to establish a minimum capability for the Project to communicate with the Data Production Software Teams. This standard should not be construed to prevent individual SCFs from implementing more advanced communications systems.

2.2.2. Standards

2.2.2.1

The SCF shall be capable of communications over the Internet. SCFs shall use the following TCP/IP based applications for Data Processing Focus Team communications:

* telnet or rlogin for establishing user sessions,
* ftp for file transfers,
* SMTP-based electronic mail clients/servers,
* talk and IRC (Internet Relay Chat) for on-line communications.

2.3. SCF Software

2.3.1. Intent

The intent of this standard is to establish uniform minimum capabilities across all SCFs. Some of these standards are intended to facilitate integration of Data Production Software into the PGS, while others are

intended to ensure a minimum ability of the Project to pass information to and from the Science Teams.

2.3.2. Standards

2.3.2.1.

SCFs hosting the PGS Toolkit shall have the following software:

* POSIX-compliant UNIX operating system supporting the XPG3 standard,
* TCP/IP communications,
* SMTP-compatible e-mail protocol,
* X-window, X11.R4,
* X-window based window manager,
* Word processor producing output file in one of the following formats: Word, Word Perfect, or PostScript.

2.3.2.2.

SCFs not hosting the PGS Toolkit shall have the following software:

* TCP/IP communications,
* SMTP-compatible e-mail protocol

2.4. SCF Security

2.4.1. Intent.

The intent of this standard is to establish minimum security requirements for networked SCFs. More elaborate security may be imposed locally.

2.4.2. Guidelines

2.4.2.1

Multi-user SCFs should use system security with procedures to establish and maintain, as a minimum:

* Passwords,
* User Accounts,
* User Permissions.

2.4.2.2.

All passwords on networked, multi-user SCFs should adhere to the following:

* Passwords should consist of 6 or more characters, including at least

one numeric or "special" character (such as a space or asterisk);
* Passwords should not be the user's login name, or circular shift of that name

2.4.2.3

A password aging scheme should be employed, to force users to change passwords periodically (e.g., every 90 days).

2.4.2.4.

The System Administrator of networked SCFs should:

* Reset all factory-set passwords immediately;
* Disable the Guest account;
* Either omit the Anonymous FTP account or restrict it to a specific directory;
* Assure the initial password for a new account is not a forename-surname or other usage of the user name;
* Minimize distribution of the root password.

2.4.2.5.

The System Administrator of networked, multi-user SCFs should:

* Establish groups for users with read and execute permissions for group members;
* Remove global read/write access to any files which are not for public view;
* Remove global execute access to any directory which is not for public access.

2.4.2.6.

.netrc should not be used (would allow login without a password).

3. Data Production Software Standards and Guidelines

3.1. C Coding Standards

3.1.1. Intent

Adherence to these standards is mandatory for data production code generated in the C language.

3.1.2. Standards.

The mandatory coding standards for data production code generated in the C language follow.

3.1.2.1

The source code shall comply with the ANSI standard specification for C (ANSI/X3.159-1989; C Programming Language Standard). Vendor specific extensions to the standard shall not be used. The rationale for this standard is portability.

3.1.2.2

External calls from the operational data production software in the PGS, for system and resource accesses, file I/O requests, error message transaction, and metadata formatting, shall be made through PGS Toolkit calls. The rationale for this standard is portability as well as having one group develop this code instead of all science teams duplicating the development effort.

3.1.2.3

(CH01) The source code shall comply with IEEE Standard 1003.1, POSIX-Part 1: System Application Program Interface (API) [C Language] and with the restrictions found in Appendix D of the PGS Toolkit Users Guide for the ECS Project.

3.1.3. Guidelines

Compilation

3.1.3.1.

It is strongly recommended that code be compiled with the ANSI checking option on. The rationale for this guideline is portability.

3.1.3.2

File Inclusion. <> and "" notation should be used for including standard C header files and programmer created header files, respectively. The rationale for this guideline is maintainability.

Declarations.

3.1.3.3.

All variables shall be initialized prior to use (except for static variables), i.e. no assumptions should be made that variables are initialized to zero. The rationale for this guideline is portability.

3.1.3.4.

(CH01) Declarations should be ordered consistently throughout the code. An example of a consistent order is:

* declaration of module arguments in the same order as the argument list (before the opening brace of the function), (CH01)
* declaration of external variables,
* declaration of local variables,
* declaration of functions used.

(CH01) The rationale for this guideline is maintainability.

3.1.3.5.

(CH01) Naming Convention. A consistent and descriptive naming convention should be adopted. The rationale for this guideline is maintainability.

Structuring

3.1.3.6

.Loop control variables should be of INTEGER type. The rationale for this guideline is portability. Note: This guideline includes the subtypes of integer, such as char, short, long, etc.

3.1.3.7.

Unconditional branching (GOTO) should only be used within nested structures and should only reference a label further down in the code. The rationale for this guideline is maintainability.

3.1.3.8.

Use a consistent style to highlight code structure and increase readability. The rationale for this guideline is maintainability. Pointers and Arrays

3.1.3.9

A pointer should have the same type as the variable it points to. The rationale for this guideline is portability.

3.1.3.10

Only static variables are guaranteed to retain their value between module calls; all other variables should be assumed to be undefined for each access of a module. The rationale for this guideline is portability.

3.1.3.11.

Contiguous use of memory by arrays should not be assumed. The rationale for this guideline is portability. Operators

3.1.3.12.

Avoid implicit type casts. The rationale for this guideline is portability. Note: C language processors will generate code to cast between the types on the left and right side of an assignment operator, but reliance on this implicit type conversion is not a good idea. The programmer should include explicit type casts:

* long integer_variable;
* long * integer_pointer;
* double floating_point_var;
* integer_pointer = &integer_variable; /* per guideline */
* (CH01) integer_pointer = &floating_point_variable; /* contrary to guideline */
* integer_variable = (long)floating_point_variable; /* per guideline */

3.1.3.13.

Float and double variables should not be compared for strict equality (i.e., using == or !=). The rationale for this guideline is portability. Functions

3.1.3.14.

All functions should be typed (and preferably prototyped). The rationale for this guideline is maintainability. Note: the use of function prototypes significantly reduces interface errors in C. A function prototype tells the compiler the type of the function and the number and types of the arguments. An example follows:

* int GreatestCommonDenominator(int large_term; int small_term);

3.1.3.15.

Functions which do not return a value should be typed as void. The rationale for this guideline is maintainability.

3.1.3.16

The div and ldiv functions in the C Standard Library should be used to obtain consistent values of remainders when the quotient is negative. The rationale for this guideline is portability, since the C language definition does not specify how to handle this situation, and at least two answers are possible. Note: in general static analysis cannot ensure that the quotient will never be negative so the library functions should be used instead of the % operator.

3.1.3.17

The correct functioning of code should not depend on the rounding behavior of converting a long double to other floating types or double to a float. The rationale for this guideline is portability.

3.2. FORTRAN Coding Standards

3.2.1. Intent

Adherence to these standards is mandatory for data production software generated in the FORTRAN language.

3.2.2. Standards.

The mandatory coding standards for data production code generated in the FORTRAN language follow.

3.2.2.1.

The FORTRAN compiler standard consists of the following parts:

* (CH01) Science data production source code shall comply with the ANSI standard specification for FORTRAN77 or for FORTRAN90.
* Heritage FORTRAN66 code shall be made to compile using FORTRAN77 or FORTRAN90 prior to delivery to the DAAC. The rationale for this standard is portability.

3.2.2.2.

External calls from the operational data production software in the PGS, for system and resource accesses, file I/O requests, error message transaction, and metadata formatting, shall be made through PGS Toolkit calls. The rationale for this standard is portability as well as having one group develop this code instead of all science teams duplicating the development effort.

3.2.2.3

(CH01) FORTRAN77 code shall not use PGS Toolkit calls that provide capabilities not found in FORTRAN77 (e.g. dynamic memory allocation). The rationale for this standard is that there will be no FORTRAN77 bindings for this class of PGS Toolkit calls.

3.2.2.4

(CH01) The FORTRAN77 source code shall comply with IEEE Standard 1003.9, POSIX FORTRAN77, Language Interfaces, Part 1: Binding for System Application

Program Interface (API) and with the restrictions found in Appendix D of the PGS Toolkit Users Guide for the ECS Project.

3.2.3. Guidelines

FORTRAN77 Extensions

3.2.3.1.

The following extensions to FORTRAN77 may be used:

* INCLUDE statement,
* BYTE data type,
* DO WHILE or EXIT,
* ENDDO,
* STRUCTURE data types,
* names up to 31 characters in length,
* IMPLICIT NONE statement,
* block IF with ELSE IF and END IF,
* in-line comments,
* extended character set to include lower case letters, underscore, left and right angle bracket, quotation mark, percent sign, and ampersand,
* initialization of data in declaration,
* long line extensions beyond 72 characters per line.

(CH01)The ESDIS Project will attempt to procure FORTRAN77 compilers for the DAACs that permit the above extensions to the ANSI standard language, but the DAAC compiler acceptance of these extensions can not be guaranteed. Those science teams possessing heritage code that uses most of these extensions may wish to consider coverting the code to use a Fortran 90 compiler. The FORTRAN77 language and several extensions are a subset of the FORTRAN90 language.

(CH01)Compatibility with the next FORTRAN Standard
Any constructs and features of the FORTRAN90 language that are marked for removal in the next release of the FORTRAN standard should not be used.

Declarations

3.2.3.2

All variables should be initialized prior to use. The rationale for this guideline is portability.

3.2.3.3

(CH01) Declarations should be ordered consistently throughout the code. An example of consistent ordering is:

* declaration of module arguments in the same order as the argument list,
* declaration of global variables in COMMON (using INCLUDE files),
* (CH01) declaration of local PARAMETERs (types and values),
* declaration of local variable types,
* declaration of user defined function types used,
* EXTERNAL declarations,
* INTRINSIC declarations,
* (CH01)declaration of DATA values.

The rationale for this guideline is maintainability.

3.2.3.4

PARAMETER variables should not be redefined. The rationale for this guideline is maintainability.

3.2.3.5

COMMON blocks should be inserted into the code using INCLUDE. The rationale for this guideline is maintainability.

3.2.3.6

(DELETED) Naming Convention

3.2.3.7

(CH01) A consistent and descriptive naming convention should be adopted. The rationale for this guideline is maintainability.

Structuring

3.2.3.8

Loop control variables should be of INTEGER type. The rationale for this guideline is maintainability.

3.2.3.9

Unconditional branching (GOTO) should only be used within nested structures. The rationale for this guideline is maintainability.

3.2.3.10

Computed and arithmetic GOTOs should not be used. The rationale for this guideline is maintainability.

3.2.3.11

DO-loops should be terminated with CONTINUE or ENDDO. The index of a DO-loop should always be of integer type, and the index should not be modified inside the loop. The rationale for this guideline is maintainability.

3.2.3.12

Use a consistent style to highlight code structure and increase readability. The rationale for this guideline is maintainability. Operators

3.2.3.13

Real and complex variables should not be compared for strict equality (i.e., using .EQ. or .NE.). The rationale for this guideline is portability.
Labeling

3.2.3.14

(CH01)A consistent labeling scheme should be used. The rationale for this guideline is maintainability.

Functions

3.2.3.15

Generic intrinsic functions should be used rather than type specific functions. The rationale for this guideline is portability.

3.2.3.16

The correct functioning of code should not depend on the rounding behavior of converting a REAL*8 or COMPLEX to other floating types. The rationale for this guideline is portability.

3.3. Ada Coding Standards

3.3.1. Intent

Adherence to these standards is mandatory for data production software written in the Ada language.

3.3.2. Standards.

The following are the list of mandatory standards for data production code generated in the Ada language:

3.3.2.1.

The source code shall comply with the ANSI standard specification for Ada (MIL-STD-1815-A, U.S. Department of Defense, Ada Language Reference Manual).

The rationale for this standard is portability.

3.3.2.2

External calls from the operational data production software in the PGS, for system and resource accesses, file I/O requests, error message transaction, and metadata formatting, shall be made through PGS Toolkit calls. The Pragma Interface Statement shall be used to interface with all commercial library software provided as part of the PGS Toolkit. Ada bindings for the PGS Toolkit will not be developed and it is the software developer's responsibility to utilize the FORTRAN or C version of the PGS Toolkit correctly. The rationale for this standard is portability as well as having one group develop this code instead of all science teams duplicating development effort.

3.3.2.3

The delivered code shall not make PGS Toolkit calls from within Ada tasks. The rationale for this standard is operability and maintainability.

3.3.2.4

There are no Ada libraries supported. All software shall be delivered in source code form.

3.3.3Guidelines

3.3.3.1

Since Chapter 13 of the Ada Language Reference Manual deals specifically with features for adapting Ada to platform-unique features, Chapter 13 features should be used with care. Such code should be isolated as much as possible and commented extensively to make the task of porting to a new host as easy as possible.

3.3.3.2

There are references for the production of good Ada code. The following reference might be useful in developing local coding guidelines and styles: Seidewitz E., et al., Ada Style Guide, Version 1.1, GSFC SEL 87-002, 1987.

3.4. Module Identification Standard.

3.4.1. Intent.

Module Identifications are needed primarily to ease maintenance and secondarily to assist in the I&T process.

3.4.2. Standard

To allow identification of individual items of code, a header (prolog), as defined below, shall be inserted at the top of each module in the production software. A module is a main program, subroutine, procedure, function, etc. This header should also be included at the top of insert/include files, with the exception that the blocks relating to input and output parameters are omitted. Note: The example uses the C language style of comments, and would need to be converted for other languages. All entries preceded with "!" are mandatory with the exception of "!Team-unique Header:" if there is no team-unique header.

For heritage code, where the generation of this header information for every module is an unreasonable burden, an alternative approach based on compilation units can be used. A single header can be used to record the description and version history for all modules contained within a file which is compiled as a unit. In this case the only requirement for each module is to describe each input/output parameter (not globals). Although this alternative approach is acceptable for heritage code, the former approach, with a header for each module, is required for all new code.

The inclusion of the headers will allow automated tools within the DAAC I&T environment to manipulate the software items and will ease understanding by the I&T team.<

Templates for module headers and include file headers can be accessed through the PGS Toolkit interface software.

```
line no.
00 geonav(float pos[3],float smat[3][3],float coef[6],float sun[3],
01 int nsta,int ninc,int npix,float xlat[409],float xlon[409],
02 float solz[409],float sola[409],float senz[409],float sena[409])
03/*
04!C*******************************************************
05
06 !Description: This subroutine calculates the sensor
07 orientation from the orbit position vector and input values of
08 attitude offset angles. The calculations assume that the angles
09 represent the yaw, roll and pitch offsets between the local
10 vertical reference frame (at the spacecraft position) and the
11 sensor frame. The outputs are the matrix which represents
12 the transformation from the geocentric rotating to sensor
13 frame, and the coefficients which represent the Earth scan
14 track in the sensor frame. The reference ellipsoid uses an
15 equatorial radius of 6378.137 km and a flattening factor of
16 1/298.257 (WGS 1984).05
17
18 !Input Parameters:
19 pos[3] satellite position
20 coef[6] scan line coefficients
```

21 sun[3] unit sun vector in geocentric rotating coordinates
22 nsta number of first pixel to start with
23 ninc increment between pixels for computations
24 npix number of pixels in scan line
25
26 !Output Parameters:
27 xlat[409] latitude values
28 xlon[409] longitude values
29 solz[409] solar zenith values
30 sola[409] solar azimuth values
31 senz[409] sensor zenith values
32 sena[409] sensor azimuth values
33
(CH01)34 input/output parameters:
35 smat[3] [3] sensor orientation matrix
36 !Revision History:
37 $LOG: geonav.c,v $
38 Revision 01.01 1993/10/12 10:12:28
39 Z. GREEN (zgreen@harp.gsfc.nasa.gov)
40 Initial delivery of software. Modified to comply with ESDIS
41 standards. It was a breeze.
42
43 Revision 01.00 1993/04/29 17:12:28
44 A. SMITH (asmith@harp.gsfc.nasa.gov)
45 Initial debugged version, based on original FORTRAN77
46 subroutine developed in 1983 by C. ADAMS of the TELLUS
47 project.
48
49 !Team-unique Header:
50 Science team puts any thing they want in this portion of prolog
51 !END**********
*/
code follows here

The following notes describe how to use the header block, using the above example as a reference.

Line 00: Name of main procedure, include file or subroutine/ procedure/ function. If this is not the main procedure or an include file, it should contain the function statement.

Line 04: Start of prolog. Initial marker can take the following values:

!FXY - contains FORTRANXY (XY = 77 or 90) executable statements
!C - contains C executable statements
!ADA - contains Ada executable statements
!FXY-INC - FORTRANXY (XY = 77 or 90) include file
!C-INC - C include file

Line 06: A concise but complete summary of the overall function of the module. Any references for methods and/or algorithms should be included. Use as many lines as necessary.

Line 18: Header for input parameters.

Line 19-24: Input parameters (not global variables) in the order they are presented to the module with a short 1-2 line description of the parameter (and its units where appropriate). Global variables should be described in the module in which they are declared (i.e., only once).

Line 26: Header for output parameters.

Line 27-32: Output parameters (not global variables) in the order they are contained in the function statement. Global variables should be described in the module in which they are declared (i.e., only once). Same format as for input parameters.

(CH01) Line 34-35: Parameters that serve both as input parameters and as output parameters (not global variables) in the order they are contained in the function statement. Global variables should be described in the module in which they are declared (i.e., only once). Same format as for input parameters. The automated I&T tools will allow this part of the header to be omitted.

Line 36: Start of Revision History Log.

Line 37: If you are using an automated tool for revision control, you should insert any statements required immediately after the Modification History Log Header.

Line 38-47: Each revision should contain as a minimum the revision number, date, time, person, and email address, with a short description of ALL the changes made. The first revision should include the original author of the code. Revisions should be ordered with the latest first. [Note: this revision information can be supplemented with more detailed comments in the code referencing the revision number.]

Any revision numbering system relevant to your site and configuration control mechanism may be used but the "nn.mm" format is recommended, where "mm" is updated each time a change is made to the module and "nn" is updated when the function of the module changes or the algorithm/method is changed. [Note: The release number for the data production software is not related to the revision numbers on individual modules - the release number scheme should be determined by the development team. The date associated with a release is more meaningful than a release number to those outside the development team.]

Line 49: Start of Team-unique Header.

(CH01) Line 50: Each team may design it's own header section(s). There may be more than one of these, mixed in with the mandatory header sections of

the prologue. The automated I&T tools will ignore this part of the header.

Line 51: End of source code prolog.

(CH01)

3.5 Script Languages Standard

3.5.1 Intent

Adherence to these standards is mandatory for script command languages that are used to generate the command language portion of the science data production software delivered to the DAACs.

3.5.2 Standard

(CH01) Command language code delivered to the DAACs as part of the science data production software shall be written using one or more of the following script languages:

* csh,
* ksh,
* perl,
* POSIX-compatible shell language.

(CH01) The POSIX standard is published in IEEE Std 1003.2-1992, IEEE Standard for Information Technology, Portable Operating System Interface (POSIX), Part 2: Shell and Utilities.

3.5.3 Guidelines

3.5.3.1

(CH01) Command language code delivered to the DAACs as part of the science data production software from a science team should be written using the smallest possible number of script languages.

3.5.3.2

(CH01) The use of compiled programming language should be maximized and the use of interpreted script command language should be minimized, in developing science data production software for delivery to the DAACs. The rationale for this guideline is efficiency of software execution and adherence to standards for portability. Compiled code executes more efficiency than interpreted code. There are formal national standards for the approved compiled programming languages while there is only one formal national standard, POSIX, for a script language.

APPENDIX B: RATIONALE FOR CODING STANDARDS AND GUIDELINES

4.2.1.1	Portability
4.2.1.2	Portability as well as having one group develop this code instead of all science teams duplicating this development effort.
4.2.1.3	Portability
4.2.1.4	Maintainability
4.2.1.5	Portability
4.2.1.6	Portability
4.2.1.7	Portability
4.2.2.1	Portability
4.2.2.2	Maintainability
4.2.2.3	Maintainability
4.2.3.1	Portability, Reliability
4.2.3.2	Maintainability
4.2.3.3	Maintainability
4.2.4.1	Portability
4.2.4.2	Portability
4.2.5.1	Portability
4.2.6.1	Maintainability
4.2.7.1	Reliability
4.3.1.1	Portability
4.3.1.2	Portability
4.3.1.3	Reliability
4.3.1.4	Portability
4.3.2.1	Maintainability
4.3.2.2	Maintainability
4.3.2.3	Portability
4.3.3.1	Maintainability
4.3.4.1	Maintainability
4.3.4.2	Portability
4.4.1.1	Portability
4.4.1.2	Portability as well as having one group develop this code instead of all science teams duplicating this development effort.
4.4.1.3	Portability
4.4.1.4	Portability
4.4.1.5	Portability
4.4.1.6	Portability
4.4.2.1	Maintainability
4.4.2.2	Portability
4.4.2.3	Maintainability
4.4.2.4	Maintainability
4.4.2.5	Maintainability
4.4.3.1	Maintainability
4.4.3.2	Maintainability
4.4.3.3	Maintainability
4.4.3.4	Maintainability

4.4.4.1	Maintainability
4.4.5.1	Portability
4.4.6.1	Maintainability, Reliability
4.5.1.1	Maintainability
4.5.1.2	Portability
4.5.1.3	Reliability
4.5.2.1	Maintainability
4.5.2.2	Maintainability
4.5.2.3	Portability
4.5.3.1	Maintainability
4.5.3.2	Maintainability
4.5.3.3	Maintainability
4.6.1.1	Portability
4.6.1.2	Portability as well as having one group develop this code instead of all science teams duplicating this development effort.
4.6.1.3	Portability
4.6.2.1	Maintainability
4.6.2.2	Portability
4.6.2.3	Maintainability
4.6.2.4	Maintainability
4.6.2.5	Maintainability
4.6.3.1	Maintainability
4.6.3.2	Maintainability
4.6.3.3	Maintainability
4.6.3.4	Maintainability
4.6.3.5	Maintainability
4.6.4.1	Maintainability
4.6.5.1	Portability
4.6.6.1	Maintainability, Reliability
4.7.1.1	Maintainability
4.7.2.1	Maintainability
4.7.3.1	Maintainability
4.7.3.2	Maintainability